AFRL-VS-PS-
TR-2004-1180

SMART MEMORY SYSTEMS: POLYMORPHOUS COMPUTING ARCHITECTURES

Mark Horowitz

Stanford University
Gates 306, 353 Serra Mall
Stanford, CA 94305-9030

31 August 2004

Final Report

AIR FORCE RESEARCH LABORATORY
Space Vehicles Directorate
3550 Aberdeen Ave SE
AIR FORCE MATERIEL COMMAND
KIRTLAND AIR FORCE BASE, NM 87117-5776

DTIC COPY

AFRL-VS-PS-TR-2004-1180

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data, does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report has been reviewed by the Public Affairs Office and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

If you change your address, wish to be removed from this mailing list, or your organization no longer employs the addressee, please notify AFRL/ VSSE, 3550 Aberdeen Ave SE, Kirtland AFB, NM 87117-5776.

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

This report has been approved for publication.

//signed//

JIM LYKE
Project Manager

//signed//

KIRT S. MOSER, DR-IV
Chief, Spacecraft Technology Division

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
31-08-2004	Final Report	01-06-2001 to 31-05-2004

4. TITLE AND SUBTITLE
SMART MEMORY SYSTEMS: Polymorphous Computing Architectures

5a. CONTRACT NUMBER
F29601-00-2-0085

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62712E

6. AUTHOR(S)
Mark Horowitz

5d. PROJECT NUMBER
DARP

5e. TASK NUMBER
SC

5f. WORK UNIT NUMBER
AF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Stanford University
Gates 306, 353 Serra Mall
Stanford, CA 94305-9030

8. PERFORMING ORGANIZATION REPORT

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research
Space Vehicles Directorate
3550 Aberdeen Ave. SE
Kirtland AFB, NM 87117-5776

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-VS-PS-TR-2004-1180

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We describe a new universal computing element for future embedded applications. Our polymorphic architecture contains course-grain reconfigurable processors, memory, and network. Each application is compiled into a set of programs for the nodes, and a set of configuration files for the chip. These configuration files optimize the chip for the kinds of tasks that the application demands, creating efficient SIMD engines for the stream-oriented portions of applications and efficient thread machines for the control-intensive portions.

To efficiently program our polymorphic architecture, we created a number of abstract machine models that efficiently implement different models of computation and then created a set of virtual machine simulators to allow software development before the hardware was present. These virtual machine interfaces also allow an embedded system to evolve as new hardware or software components are added since they represent a stable, parameterized abstract interface between the hardware and the software.

15. SUBJECT TERMS
Space Vehicles; Smart Memory; Polymorphic Computing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	Unlimited	42	(505) 846-5812

1. Introduction

In our prior work we demonstrated both the effectiveness of a stream programming model (Imagine) and the utility of a polymorphic architecture that can execute streams and threads effectively (Smart Memories). The goal of this research effort was to extend that work to create a usable polymorphic computing architecture and to create low-level middleware that together would form a foundation for future embedded systems. We accomplished this goal with the creation of the Smart Memory chip architecture – a polymorphous computing base that can support both stream and thread computation efficiently – and creating the core virtual machine abstractions to present stable programming interfaces. The two virtual machine interfaces, the Stream Virtual Machine (SVM) and the Thread Virtual Machine (TVM) are still being refined in the morphware forum, but the basic principles have been laid out.

The continued scaling of integrated circuit fabrication technology has driven much of the information technology revolution. The uses of computation have exploded as the costs have continued to scale. Yet this same scaling has also had an effect on the architecture of future computing systems, and these factors will only grow with time. Scaling makes computation cheaper, smaller, and lower-power, thus enabling more sophisticated algorithms in a growing number of embedded applications. This spread of low-cost, low-power computing can easily be seen in today's wired (e.g. gigabit ethernet or DSL) and wireless communication devices, gaming consoles, and handheld PDAs. These new applications have different characteristics from today's standard workloads, often containing highly data-parallel streaming behavior. While the applications will demand ever-growing compute performance; power (ops/W) and computational efficiency (ops/$) are also paramount; therefore, designers have created narrowly-focused custom silicon solutions to meet these needs.

The scaling of process technologies, however, makes the construction of custom solutions increasingly difficult due to the increasing complexity of the desired devices. While designer productivity has improved over time, and technologies like system-on-a-chip help to manage complexity, each generation of complex machines is more expensive to design than the previous one. High non-recurring fabrication costs (e.g. mask generation) and long chip manufacturing delays mean that designs must be all the more carefully validated, further increasing the design costs. Thus, these large complex chips are only cost-effective if they can be sold in large volumes. This need for a large market runs counter to the drive for efficient, narrowly-focused, custom hardware solutions.

As we said in our proposal, there is a need to create a new universal computing element for future embedded applications, much as the microprocessor has served as a universal computing element for the last thirty years. Our polymorphic architecture uses application-specific configuration to achieve much higher performance and power efficiency than conventional processors. At the core of this research is the creation of a Smart Memory architecture, which contains reconfigurable processors, memory, and network. Each application is compiled into a set of programs for the nodes, and a set of configuration files for

the chip. These configuration files optimize the chip for the kinds of tasks that the application demands, creating efficient SIMD engines for the stream-oriented portions of applications, and efficient thread machines for the control-intensive portions.

To efficiently program our polymorphic architecture, we created a number of abstract machine models that efficiently implement different models of computation, and then created a set of virtual machine simulators to allow software development before the hardware was present. These virtual machine interfaces also allow an embedded system to evolve as new hardware or software components are added, since they represent a stable, parameterized abstract interface between the hardware and the software.

To support the streaming portions of applications, we created the Brook programming language. Brook is a set of extensions to C++, and builds on our prior work with stream programming on Imagine. In fact the Imagine substrate was essential for this work to proceed. During this contract we also completed the Imagine development, and built and debugged a number of Imagine demonstration boards. We used these systems to debug a number of stream programming concepts.

While streaming is a powerful computation model, it is not universal. To support the non-streaming portions of applications, we look at ways to extend the thread model to parallelize applications more easily. In particular we worked on using speculation to allow a compiler to perform optimizations that it was pretty sure were useful, but could not prove were safe. The hardware buffering would allow the application to back out of bad decisions if needed. This system was tested with both standard C/Fortran benchmarks, as well as a number of JAVA applications.

In addition to the work on programming abstractions, we felt that software robustness was a critical issue for these large embedded systems, and created a research program to investigate using a meta-compiler that can learn and check assertions about the software while it is compiling code. While this work is still in progress, the results have been impressive. The system we developed can handle millions of lines of code, and has found a number of serious errors in production code. This technology was spun out into a commercial startup.

Driving the development of our polymorphic architecture and its software is a set of applications that address both the streaming and threaded applications, with emphasis on the streaming application, since threaded applications are more common.

Our work was divided into five broad research areas, which are summarized in the following sections of the report. More information about all this work can be found in the publications that are also attached to this report. The next section first describes the programming models that we think are critical for future systems, and formed the bases of much of the PCA program. Section 3 then describes the hardware work done under this contract, which included work on both Smart Memories and Imagine. Section 4 then describes the

computational abstractions we created for this machine, and the virtual machines that were created to support these abstractions. Section 5 describes our programming language support, for more robust software design. Section 6 reviews the applications work done as part of the contract.

2. Programming Models and Software Tools

Before describing our hardware and middleware efforts in more details, we first look at the programming models we felt our hardware needed to support. We focused on creating programming models that made it easier for the programmer to exploit the large available parallelism of future VLSI computing chips. This led us to work primarily on streams, and speculative threads.

2.1. Stream Programming

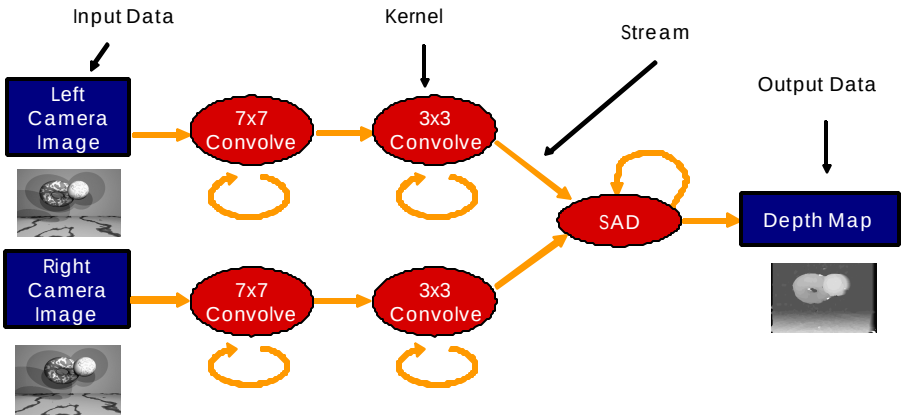

Figure 2 : Stream and kernel diagram for stereo depth extractor application

The stream programming model represents applications as a set of computation kernels that consume and produce data streams. Each data stream is a sequence of data records of the same type, and each kernel is a program that performs the same set of operations on each input stream element, and produces one or more output streams. Many languages have been developed to support this type of program. For Imagine, applications are programmed using this model with two languages: *StreamC* and *KernelC*.

A StreamC program specifies the order of kernel executions and organizes data into sequential streams that are passed from one kernel to the next. For example, Figure 2 shows a graphical representation of the StreamC program for a stereo depth extractor application [1]. This application processes images from two cameras that are positioned at a horizontal offset from each other. The images are first pre-processed by two convolution kernels (3x3 convolve and 7x7 convolve). Then the Sum-of-Absolute-Differences (SAD) kernel is called repeatedly to find the number of pixels of horizontal shift that minimizes the SAD of a 7x7 area centered around each pixel. Each call to the kernel searches a different set of horizontal offsets, or disparities. The depth at each pixel is proportional to the inverse of the best disparity match. Notice, in the example above, that data in each output stream is destined either for the next

kernel in the application pipeline, or is consumed by the next call to the same kernel. Furthermore, though not shown in the diagram, the programmer can embed arbitrary C/C++ in a StreamC program.
Kernels in the stream programming model are programmed in KernelC. They are structured as loops that process element(s) from each input stream and generate output(s) for each output stream during each loop iteration. KernelC disallows any external data accesses besides input stream reads, output stream writes, and a few scalar parameter inputs. Kernels are also written so that successive iterations of the main loop can be executed concurrently using parallel hardware.
Mapping applications to StreamC and KernelC exposes the available parallelism and locality. Producer-consumer locality between subsequent kernels is exposed as the streams passing between kernels in StreamC. Locality within kernels is captured in KernelC. Data-level parallelism is exposed by both StreamC and KernelC because kernels perform the same computation on all of the elements of an input stream.

The Imagine software system provides the compile-time and run-time support necessary for running stream programs. As shown in Figure 3, the software system includes compilers for converting StreamC and KernelC programs into host CPU assembly code and kernel microcode, respectively, and provides runtime support for issuing stream instructions to Imagine via the host CPU's external memory interface.

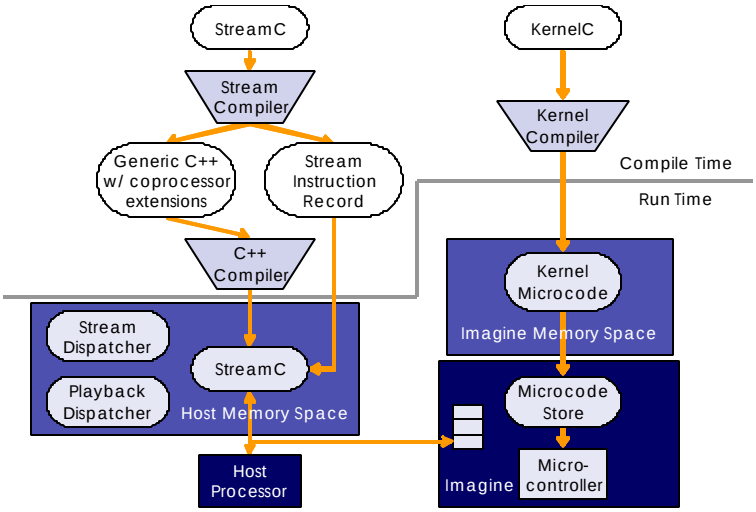

Figure 3 : Imagine software system

As part of this research effort, we have been working to create a more programmer friendly interface to streams. This work led to the creation of Brook, and to the development of the Stream Virtual Machine that will be described later in the report.

2.2. Light Weight Threads, and Thread-level Speculation

While streams are a powerful programming model, they fit best with applications with statically analyzable data flows. Many applications have large amounts of data parallelism, but with more dynamic communication patterns. The natural programming model for these applications is to have large number of light-weight threads, and then context switch out threads that are waiting for communication. As we will describe later in the hardware section, the programmable memory has a number of features that allow very efficient synchronization, and fast wake-up of stalled processors.

We also worked on extending this model to application where you thought there was high probability of parallelism, but where the compiler could not guarantee it. We had previously shown that using thread-level speculation could help these applications. This is where hardware is added to allow a processor to speculatively execute some code, and then rollback the results if an unexpected communication did occur. The programmable memory on a Smart Memory chip also made this type of operation possible. To make creating these applications easier, we have investigated using thread-level speculation to make writing parallel applications to run on the SM architecture almost as easy as writing sequential applications. To achieve this goal we developed a complete system for dynamic detection of parallelism and recompilation into parallel speculative threads. In addition to thread-Level Speculation (TLS) hardware, we leverage a dynamically compiled language (specifically Java), and additional hardware to support low-overhead profiling. In this system, the program is annotated and run as a sequential program. The annotations and the hardware support identify the best parallel regions to be recompiled dynamically into speculative threads. Simulations indicate additional hardware support is moderate (<2% per Smart Memories quad), and programs incur a 5-15% slowdown during profiling. Our experiments have shown this system to effectively identify optimal parallel regions. Our hardware support is described in a paper presented at CGO titled "TEST: A Tracer for Extracting Speculative Threads".

To make use of the TEST support we completed the development of a dynamic parallelizing compiler for Java. This compiler takes regions selected during TEST profiling and converts them dynamically into speculative threads. Development of this compiler has yielded insights into local compiler optimizations that improve TLS performance. Optimizations include improved coding of speculative handlers and special instructions to reduce TLS overheads, register-allocated loop inductors for TLS, global register-allocation for TLS, automatic insertion of synchronization to minimize violations, and hoisting of handlers to reduce thread startup overheads. Interactions of TLS with the virtual machine that have limited speedup have led us to also develop a parallel allocator and speculative object locks.

An extensive collection of integer, floating-point and multimedia benchmarks, including many from the JavaGrande, SPECjvm98, jBYTEmark suites, have been run on simulations of this system, with encouraging speedups on a single Smart Memories quad. Speedups of 3 to 4 are seen on floating-point, 2 to 3 on multimedia and 1.5 to 2.5 on integer benchmarks. This work

was reported in an ISCA paper titled “The Jrpm System for Dynamically Parallelizing Java Programs.” This paper was selected by IEEE Micro Magazine as a “Top Pick of 2003” which recognized it as an innovative idea that will have a significant industrial impact.

3. HARDWARE

The hardware research focused primarily on the design of the Smart Memory Architecture which is described next. It consists of a flexible computing tile that can be configured to support stream and thread computation. In addition, we also worked on completing a previous generation stream machine, Imagine, to demonstrate the capabilities of this class of architecture.

3.1. Smart Memories Hardware

Smart Memories is a reconfigurable modular architecture designed to efficiently utilize resources provided by modern VLSI technology. The key insight from our prior research is that utilization of parallelism in application is essential to both high-performance and high-efficiency. In order to exploit parallelism in different classes of applications Smart Memories architecture allows reconfiguration of both processor and memory system [2].

Depending on the class of application Smart Memories chip may be configured to work as a traditional multi-processor with cache coherent shared memory, or as a multi-processor with transactional memory support, or as a streaming machine conceptually similar to Stanford Imagine processor. Within each of these modes many parts of the system can be optimized for particular application, for example, parameters of first level instruction and data cache memories.

Floorplan of Smart Memories chip is shown in Figure 3. Area of the chip is divided into tiles; each tile has processor, SRAM units, and interconnect (Figure 4). Four tiles are grouped into quads which share on-chip network interface and cache controller. Tile edge is approximately 2.5 mm in 0.1 μm technology. Relatively small tile size limits wire length and simplifies physical design. Target frequency for tile processor is 1 GHz in a commodity 0.1 μm process. This corresponds to 20 FO4 processor clock cycle while memory part of the tile works at 10 FO4 clock cycle providing double bandwidth.

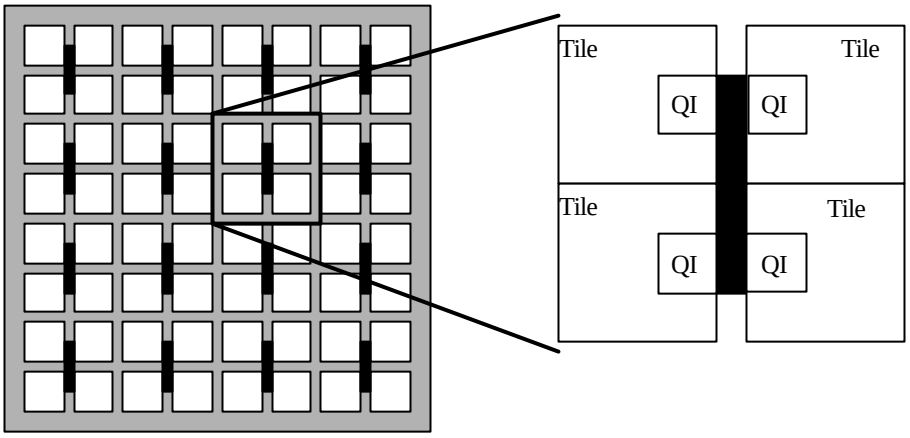

Figure 3. Smart Memories chip floorplan.

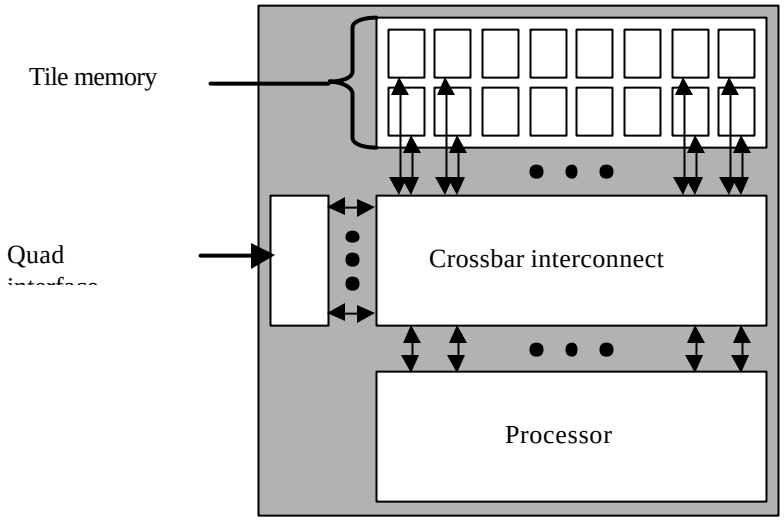

Figure 4. Tile.

Inside tile processor is connected to tile memory through crossbar interconnect which also connects quad interface to memory. Tile memory is organized as 16 independent phase-pipelined 8 KB memory mats (Figure 5). Each memory mat can read or write one 64 bit data word plus 5 bits of metadata every clock phase. In addition to decoder, data array and I/O every memory mat has storage for metadata, reconfigurable read-modify-write logic that can atomically perform logical operations on metadata, head/tail pointers and strides for hardware FIFOs, configurable comparator for cache tags, write buffer.

To demonstrate possibility of an efficient implementation of memory mat test chip with 2KB 32-bit wide memory mats has been designed and manufactured in a TSMC 0.18 µm technology [3]. At the nominal 1.8V supply, the testchip operates up to 1.1GHz, a 10FO4 cycle. The power and area of the peripheral circuits as a percentage of the mat are 26% and 32% respectively. The cell area efficiency of the mat is 61%. Thus for a modest hardware overhead, we have added reconfigurability to a basic SRAM array. We trade off this overhead for the architectural benefit of being able to tailor the memory system to specific applications.

As shown in Figure 5 memory mats are interconnected through inter-mat control network (IMCN). IMCN is necessary when tile mats configured as cache memory. In such case some memory mats will work as cache tag storage while others are used to store cache data. Comparators in tag mats are used for tag comparison, tag mat metadata bits are used to store cache line control state, e.g. to specify whether cache line is valid, dirty etc. Result of tag and metadata comparison is sent through IMCN to data mats to enable or disable data mat operations. Write buffers in data mats are used to delay writes until result of tag comparison is received.

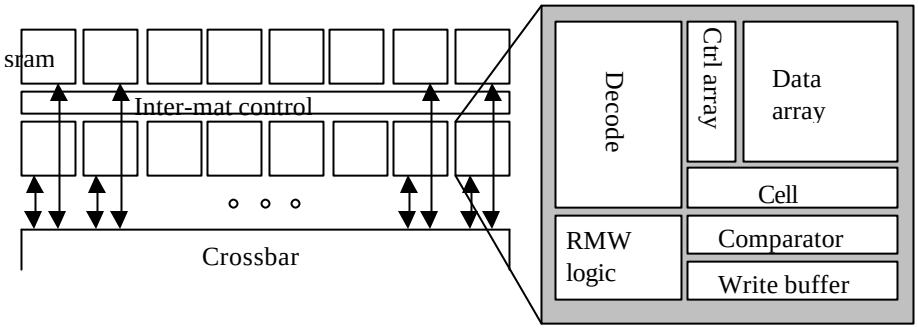

Figure 5. Tile memory system.

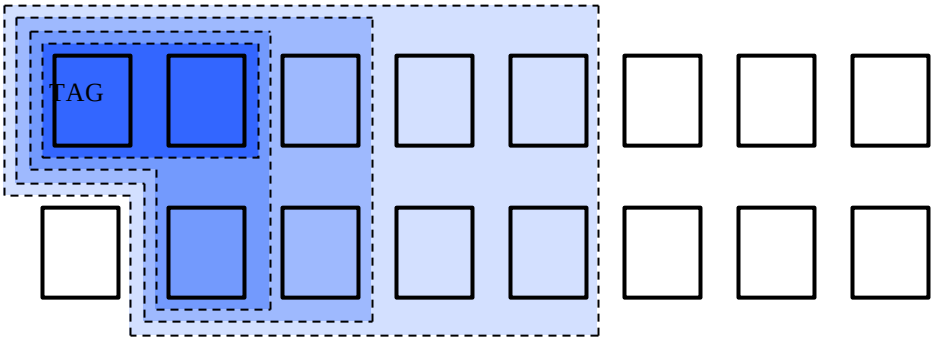

Figure 6. Cache configurations with one tag mat.

As an example Figure 6 shows all cache configurations with one tag mat. All supported cache configurations are listed in Table 1.

Table 1. Supported cache configurations.

Cache size	Number of ways	Line size, bytes	Number of used mats
8K	1	8,16,32,64	2
16K	1, 2	8,16,32,64,128	3, 4
32K	1, 2, 4	8,16,32,64,128,256	5,6,8
64K	1, 2, 4	16,32,64,128,256	9,10,12

Tile memory mats can be configured to implement 1 or 2 caches, or local scratchpad memory, or FIFO(s), or a combination of these different types of memories. Tile crossbar and processor load/store unit (LSU) as well as quad cache controller (CC) must be configured appropriately to send/route memory operations to the right memory mat or mats according to the operation address and type. In addition tile crossbar performs arbitration between different units trying to access memory mats. In the case of conflict one of access requests fails and requesting unit must reissue it.

Metadata bits associated with each word inside memory mat are used differently depending on the type of memory. Metadata inside tag mats is used to store cache line state, i.e. whether cache line is invalid, shared, exclusive or modified. Memory operations sent to cache can change state of cache line atomically using reconfigurable read-modify-write logic. For example, store that hits in the cache would change state of cache line from exclusive to modified state. Similarly cache controller can do cache line hit-invalidate as a single tag mat operation if another tile or quad has requested cache line upgrade from shared to exclusive state. Such atomic tag-compare-state-read-modify-write capabilities greatly simplify implementation of cache coherence protocol in Smart Memories.

Metadata bits in data mats or in local scratchpad mats can be used for fine-grain synchronization between threads running on the same or different tiles or quads. One of metadata bits is used as full/empty bit (F/E bit) which serves as a lock associated with word of the memory. For synchronization thread executes special memory operations – synchronized load or store. Synchronized load checks F/E bit of memory word and if F/E bit is 0 stalls the thread until F/E bit becomes 1. If F/E bit is 1, then synchronized load reads value of memory word to processor register and resets F/E bit to 0. Synchronized store stalls until F/E bit becomes 0, writes data word and sets F/E bit to 1. Such synchronized operations are used for very fast fine-grain consumer-producer synchronization in multi-threaded applications.

To support synchronized memory operations Smart Memories architecture utilizes 2 metadata bits: one is used for full/empty bit itself, second bit is waiting bit (W bit). F/E bit is flipped by read-modify-write logic inside memory mat when synchronized operation succeeds. W bit is also atomically set by memory mat whenever synchronized operation stalls indicating that there is a thread waiting on this location. When LSU receives reply from memory mat which

indicates that synchronized load or store should be stalled it stalls processor and sends a special message to cache controller. Cache controller keeps track of all outstanding memory operations originated in the quad including stalled synchronized operations. When another processor performs synchronized operation which flips F/E bit, W bit is also checked. If W bit is set, it is reset by memory mat and LSU of that processor sends a message to cache controller to wake up waiting thread. This protocol guarantees that threads stalled because of synchronized memory operations gets unstalled when another thread flips F/E bit.

Another usage of metadata bits is implementation of transactional memory. In this case thread running on tile processor might be executing speculative transaction, i.e. it may have to be squashed and all its stores may have to be discarded because of dependency violation. In order to support such functionality memory system must be able to do 2 things: to buffer all speculative stores until transaction commits and to be able to discard all speculative stores if violation is detected; to keep track of all potential dependency violations, i.e. all addresses which were read by transaction. One metadata bit in both data and tags mat is used as speculatively modified bit (SM bit) to indicate that word and cache line was speculatively written. Another metadata bit in both data and tag mats is used as speculatively read bit (SR bit) to indicate that word and cache line was speculatively read. When another less speculative or non-speculative transaction sends write SR bit for the word is checked and if it is set then dependency violation is detected and violated transaction must be interrupted. Violation interrupt handler invalidates all cache lines with SM bit set, resets all SM and SR bits and restarts speculative thread. At commit thread simply resets all SM and SR bits and starts execution of another transaction.

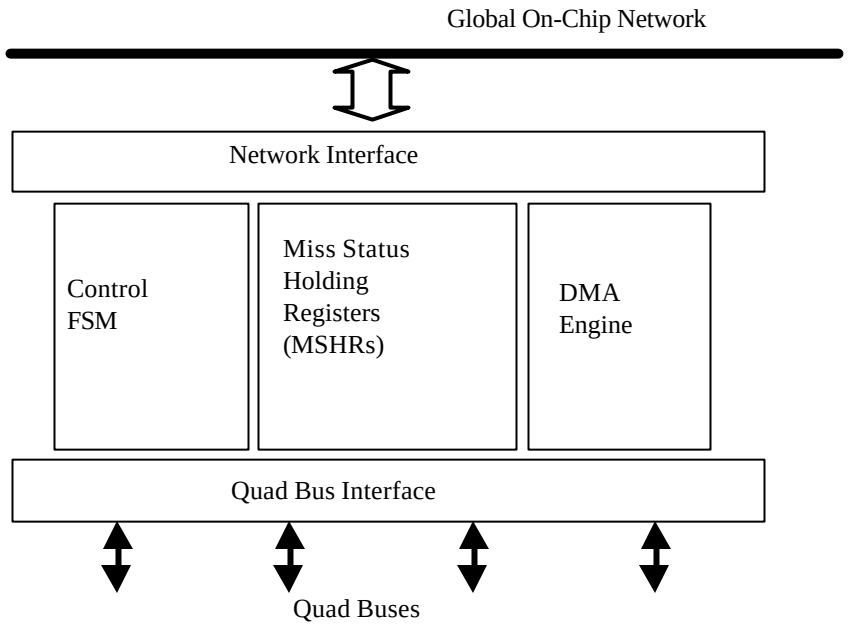

Figure 7. Cache controller block diagram.

Whenever load/store unit cannot complete memory operation immediately because of cache or synchronization miss, it sends miss message to the quad cache controller which keeps track of all outstanding memory operations originated in the quad. At the end of processing of outstanding operation cache controller completes memory operation, i.e. changes the state of memory mats and sends an acknowledgement back to LSU. Block diagram of cache controller is shown in Figure 7. Information about outstanding memory operations is stored in miss status holding registers (MSHRs). Control FSM updates state of outstanding operation in MSHR and initiates actions required to handle cache or synchronization miss. Quad bus interface receives incoming miss messages from load/store units of the tiles and performs all actions on caches in the quad such as cache line invalidation, cache line tag and data reads and writes, etc.

DMA engine is used in streaming mode to transfer streams of data between memory mats in the quad and other quads or off-chip memory. It can perform data gather to quad memory mats or data scatters from quad memory mats. Addresses for these operations can be specified as base address with stride or as an array of indices in another memory mat.

Network interface sends and receives messages to other quads and to memory controllers whenever memory operation requires interaction with external world. In addition network interface does message routing, buffering and flow control functions for global on-chip

network connecting all quad and memory controllers in Smart Memories chip. On-chip network can have very high bandwidth because of abundance of wiring resources, e.g. edge of quad can fit 16 128-bit buses. However, wire congestion limits the number of buses to 6 per quad edge. This allows quad to have one-hop and two-hop links as shown in Figure 8. Two-hop links reduce the number of hops required to route a message from one quad to another, and thus reduce network latency and increase effective bandwidth.

Each bus is double pumped, i.e. it can do 2 transfers per processor clock. This results in peak bandwidth in and out of quad equal to 1.8 Tbps.

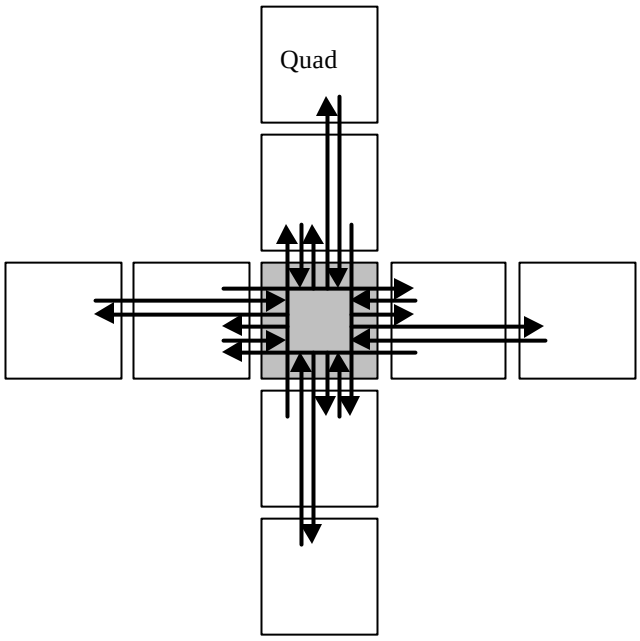

Figure 8. Inter-quad links.

Memory controllers are placed at the edges of Smart Memories chip. Memory controllers as well as off-chip DRAMs are address-interleaved based on cache line address, i.e. for each cache line there is only one memory controller which reads and writes this cache line to off-chip memory. Also the same memory controller serves as a serialization point for all cache coherence requests for this cache line and handles inter-quad cache coherence protocol. This organization spreads coherence serialization points over the chip reducing possibility of congestions.

Tile processor represents computational part of Smart Memories architecture. Similarly to memory system tile processor is reconfigurable and has several execution modes which are used for different computational models. Processor execution modes are summarized in Table 2.

Table 2. Processor execution mode summary

Mode	Instruction Encoding	Instruction width, bits	Operations per cycle	Targeted Parallelism
VLIW	RISC	128	2 integer + 2 FP	ILP
Dual-thread (single context)	RISC	32	1 integer + 1 FP each thread	Course-grain TLP
Dual-thread (multi-context)	RISC	32	1 integer + 1 FP each thread	Fine-grain TLP
Microcode (SIMD)	Microcode	256	up to 10	ILP, Data

Depending on the mode tile processor can work either as a single wide-issue processor or as 2 independent dual-issue processors. In VLIW mode processor executes a single VLIW instruction which may have up to 2 floating point and up to 2 integer operations. VLIW mode is useful for applications which exhibit a lot of instruction-level parallelism (ILP).

Dual-thread mode is intended for applications with a lot of thread-level parallelism (TLP). In dual-thread mode tile processor works as 2 independent processors, each can executes up to 2 instructions per cycle: 1 integer and 1 floating point. Furthermore, in dual-thread mode each half of tile processor can work as multiple context processor, i.e. each half has 4 hardware contexts used by different threads. When thread encounters cache misses and must be stalled because of long latency off-chip memory, half processor can quickly switch to a thread in another hardware context and continue executing useful instructions rather than wasting cycles waiting for memory.

Microcode execution mode is used for streaming applications which have a lot of data level parallelism as well as significant ILP. In this mode all 4 tiles in the quad work in lockstep as a single SIMD engine controlled by wide 256-bit microcode instruction. Microcode instruction has a separate field to control each functional unit and register file in the datapath and can encode up to 10 operations. In this mode each tile fetches only one quarter of 256-bit instructions and broadcasts it to all other tiles. Each tile assembles 256 bits microcode instruction and executes it. In microcode mode tile processor only reads and writes data into tile memory mats and communicates with other tiles over quad buses using special commands. All information required for computation must be prefetched in advance by DMAs from off-chip memory or other quads into the local memory mats.

Figure 9 shows block diagram of tile processor. It consists of memory interface or load/store unit that sends commands and messages to and receives data from memory mats and cache controller, instruction fetch unit, decode/issue unit, and datapath with 2 clusters.

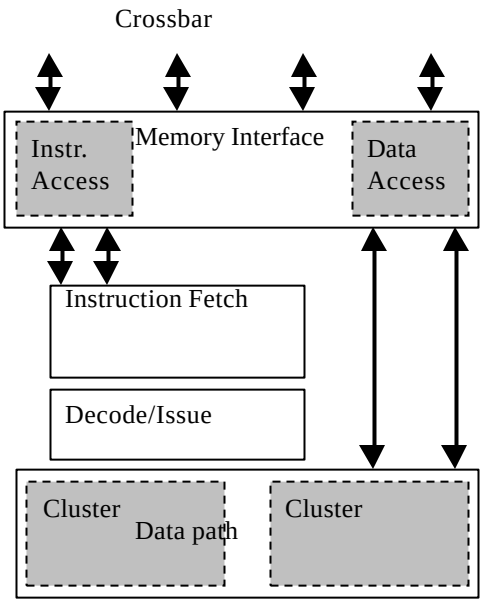

Figure 9. Block diagram of tile processor.

Instruction fetch unit is shown in Figure 10. It has 2 subunits which operate independently in multithreaded modes and combined together for wide-issue modes (VLIW and microcode). Each subunit has PC generation and context selection logic as well as state storage for 4 hardware contexts: small instruction FIFO buffer, context program counter (PC) and thread state (TS) registers. PC and instruction buffers are replicated to achieve fast context switching. Instruction fetch unit tries to prefetch instructions for contexts ready to execute. When context switch is requested instructions are immediately issued from instruction buffer and instruction fetch unit starts fetching for new active context.

Instruction fetch unit also has 2 fully-associative branch target buffers (BTB), each with 32 entries. They function independently in multithreaded modes and combined together in wide-issue modes to improve prediction accuracy.

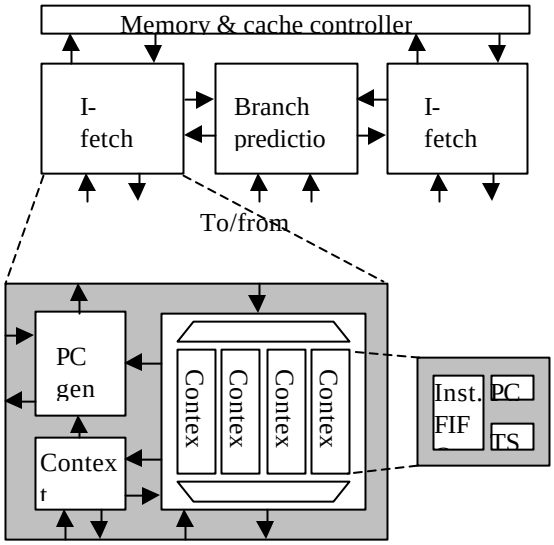

Figure 10. Instruction fetch unit.

Decode/issue unit is shown in Figure 11. It has 4 decoders for RISC instructions that translate instructions into functional unit and register file control signals in VLIW and dual-thread modes. Decode/issue unit also performs logical to physical register mapping. Register mapping is required for VLIW and dual-thread modes because the same physical registers are used differently. If multiple hardware contexts are used register mapping also depends on the currently active context.

Microcode instruction explicitly encodes all control signals for all functional units and register files. Thus instruction decoders and register mapping logic are disabled in microcode mode.

After register mapping decoded instructions are sent into pipeline registers that keep track of instructions currently executed by the processor. Bypass and stall logic checks for dependencies between instructions in the pipeline, stalls instructions if necessary, and sends control signals to datapath multiplexers if result of instruction has to be bypassed to the following instruction.

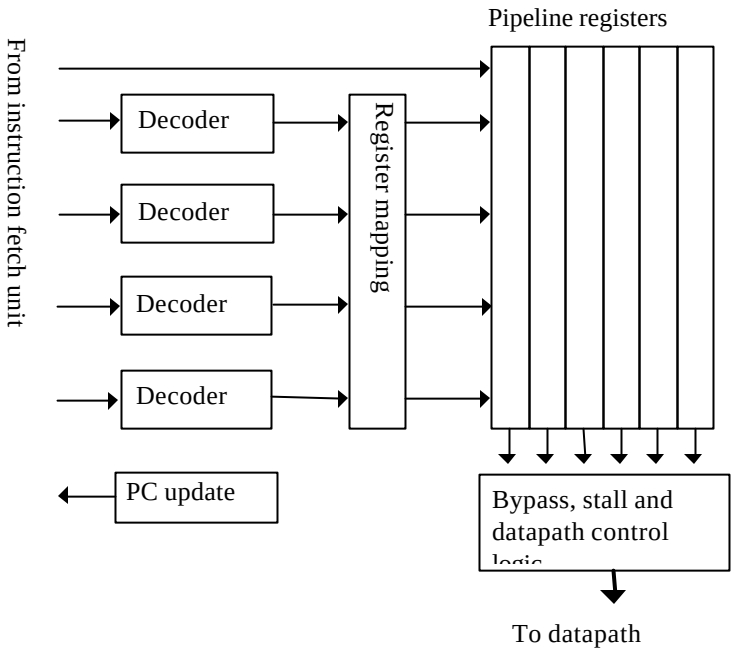

Figure 11. Decode/issue unit.

Processor datapath (Figure 12) has 2 symmetrical clusters. Each cluster contains 4 register files and 3 functional units: ALU/load/store/branch unit, floating point adder that also can perform integer ALU operations, and multiplier that can execute integer and floating point multiplication as well as division. The only part which is shared between clusters is reciprocal approximation ROM used to lookup initial approximation of reciprocal for multiplicative division instructions. Each cluster has its own path to memory interface units and can perform one load or store instruction per cycle.

In VLIW mode both clusters are used to execute instructions from the same thread. Register files in both clusters are mirrored, i.e. the result of computation in one cluster is sent to register files in both clusters. Result buses shown at the bottom of Figure 12 are used to broadcast results of computation to the whole datapath.

In dual-thread mode clusters execute instructions independently and do not share computation results.

In microcode mode each register file in the datapath is used as local register file (LRF) similar to Stanford Imagine architecture. Compiler can explicitly control register file reads and writes by microcode instruction. It must explicitly schedule all movements of values between register

file and functional units. Microcode instruction can encode up to 6 arithmetic operations and up to 4 memory operations achieving high resource utilization.

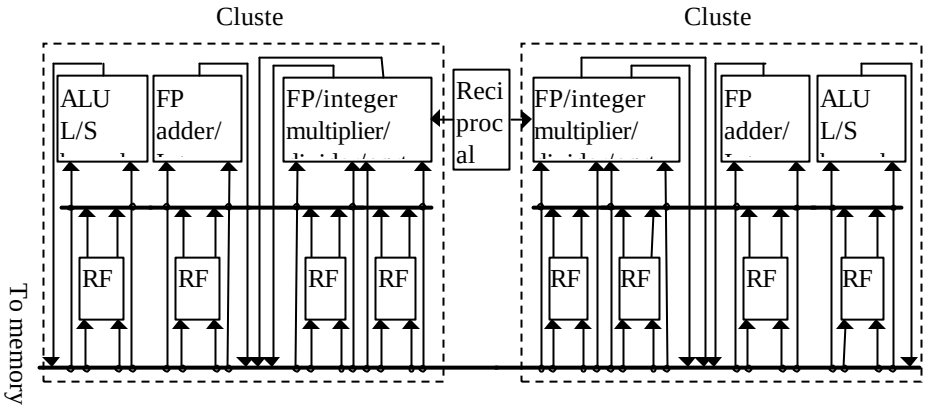

Figure 12. Datapath.

3.2. Imagine Hardware

Imagine [4] is a programmable stream processor aimed at media applications. Expressing an application as a stream program, sequences of records flowing through computation *kernels* exposes both parallelism and locality. Imagine exploits the parallelism of a stream program with an array of 48 32-bit floating-point units. Two levels of register files, 9.7 KBytes of local register files and 128 KBytes of stream register file, capture the locality of stream programs, enabling a high ratio of arithmetic to off-chip bandwidth. By keeping most data transfers local (over 95% of all transfers are from local registers) Imagine offers efficiency approaching that of an ASIC while retaining the flexibility of a programmable processor. While Imagine was designed on a previous contract, the board building/debugging and redesign was supported on this contract to help better explore stream programming and its capabilities.

The prototype Imagine processor with 21M transistors fabricated in an 0.18 µm CMOS process is designed, fabricated and running on the development board. In the section on stream programming languages, StreamC and KernelC will be described as well as compile-time and run-time systems.

3.2.1. The Imagine Development Board

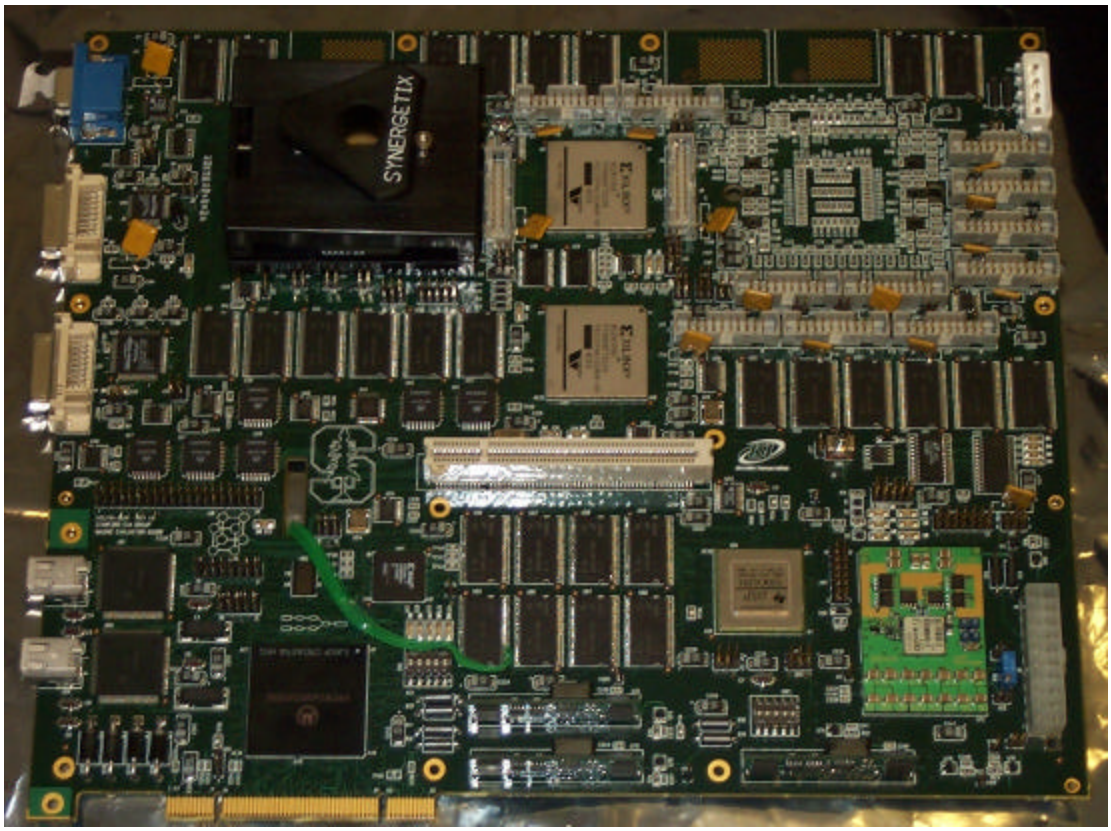

Figure 13. The Imagine development board provides a platform for testing the Imagine stream processing hardware and software tools. The first version of the Imagine development board contains two Imagine processors controlled by a single PowerPC 8240 host processor. The Imagine processors are memory-mapped to the host processor through host interface and an FPGA serves as a bridge in-between by converting their interfaces in both directions. The board is connected to a host PC with a PCI interface, providing a command-line user interface and file storage system for the host processor on the board. It also has components for I/O and a multi-processor network.

Even though the Imagine processor was functioning in the first version of the Imagine board, its huge size and low achieved host interface bandwidth induced the development of the new Imagine board. The new Imagine board contains two Imagine stream processors, two host processors, firewire, DVI output, and a multi-processor network on a full-size PCI card. Each Imagine processor is connected to a FPGA in which embedded PowerPC is used as a

dedicated host processor to the Imagine for higher achievable host interface bandwidth. Standard sized PCI card helps the board to be more easily deployed. The new-board was designed at Stanford and ISI East of University of Southern California.

4. New Virtual Machine Models

We helped design the basic model for the PCA software compilation model through our participation in the Morphware Forum. There, we helped specify a two-level compilation model with language-specific but architecture-neutral compilers in the top level and language-neutral but architecture-specific compilers in the bottom level. Connecting these two layers together are modified versions of C which we call the Thread Virtual Machine (TVM) and Streaming Virtual Machine (SVM). To define these code models, we participated in the development of specifications for the two models.

4.1. Thread Virtual Machine (TVM)

Stanford made a key contribution to the design of the TVM by developing the "hardware abstraction layer," or HAL, low-level interface. C already provides a fairly effective way to express threaded code, so we just needed to provide the enhancements necessary to patch "holes" where it cannot produce the necessary code. This is necessary in order to ensure that programmers can specify virtually *any* kind of threaded code, *all* in an architecture-neutral manner, while still providing the hooks necessary for high performance. Put another way, TVM-SVM code needs to be able to tell the low-level compiler when to emit *any* machine instruction that might be needed, and always in as architecture-neutral a way as possible. Most of C's shortcomings are in the area of accessing low-level system resources in OS or runtime system code.

One way to avoid this problem would be to provide a fixed runtime library. Library routines could be provided that would encapsulate all low-level functions and could simply be called from C routines. However, making particular library/runtime functions "native" to the TVM-SVM code model will probably overspecialize it to work well only with one particular runtime environment. One of the hallmarks of PCA architectures is their ability to morph into different configurations, many of which require fundamentally different runtime and system support. Having the compiler lock users into one runtime environment or another as a function of the compilation environment itself will probably limit wide acceptance of the system and force many programmers to seek more flexible solutions.

As a result, we decided that the TVM specification should *not* try to encapsulate larger-scale functions that are usually handled by library or runtime system code. Instead, these functions should continue to be in libraries. Moreover, these libraries should be written using the portable TVM code itself, so that they are portable from one architecture to another. There is a very good reason for this distinct division of labor: *anything* contained within the TVM specification will need to be interpreted and subsequently generated by the low-level compiler. Packing extraneous library/runtime functions into the TVM (or SVM) would therefore make the low-level compilers more complex, since they must be able to compile *all* TVM-SVM code to ensure portability. In the process, this would make building low-level

compilers for new architectures more difficult, limiting the ease with which the PCA compilation techniques could spread.

C is the baseline language for both the TVM and SVM specifications. In fact, a normal C-language program is also a simple TVM one. One of the benefits of C is that it has always been a sort of "high level assembly language." It *is* a high level language, but it is very simple and close enough "to the metal" that experienced assembly language programmers can guess fairly well how code written using C will be converted into machine instructions, much more so than with many other high level languages. It also offers direct access through operators to most of the more exotic machine instructions, such as bitwise logic and a variety of integer lengths, that are present on virtually all general-purpose CPUs today, yet are unsupported by many other languages. As a result, C is generally a good (although sometimes not ideal) choice for writing most parts of both user code and the library/runtime code that supports it. This is the model taken by many leading operating systems in order to make most of their code portable from one architecture to another, and is an example that we continue to follow.

Unfortunately, C alone cannot describe all potential OS or runtime code. All architectures possess specialized instructions used by operating system code to control the hardware at a low level in order to control the exception model for the machine, manage memory, synchronize processors, and handle I/O. Most operating systems written in C access these instructions by calling specialized assembly-language routines or through the use of inline assembly-language instructions. While the resulting code looks very different from one architecture to the next, there are some generalized functions that any of this code performs. By listing these fundamental functions and creating a standard set of macros (and/or *very* small functions) for their use, we created a "hardware abstraction layer" (HAL) for the core functions required by any OS or runtime environment. Our main effort in the design of the TVM was to lay out and specify this low-level interface, an effort that has resulted in a Morphware Forum specification document describing the 17 compiler directives and approximately 150 macros that make up the HAL.

With the HAL extensions, we can now use high-level compilers to generate TVM code for both user threads *and* most low-level OS/runtime code, and then use the same low-level compiler to compile both sets of code. The main difference between "user" and "OS/runtime" code, in fact, is that the user code accesses virtually all machine resources through system calls to an OS support library, while the OS code uses HAL primitives to directly control the machine. Since most HAL primitives will trigger privileged instruction exceptions if used in user code, this distinction is always enforced at runtime. A convenient side effect of this compilation framework is that threaded code run "on the metal," without an OS, is just TVM code that performs an application itself (like normal user code), while controlling the underlying hardware directly using privileged HAL primitives (like runtime/OS code). While some applications that target PCA architectures may be written in this manner, most will continue to use at least a minimal OS that can wrap a nice interface and important features such as error checking and memory protection around the very "raw" HAL primitives defined

here. In particular, we have worked with the TRIPS team at UT Austin to adapt IBM's K42 OS kernel to this model in order to provide an initial, UNIX-like test case. These development efforts should both lead to PCA OS environments that are fairly portable from one architecture to the next and to verify the HAL.

4.2. Stream Virtual Machine

A stream program consists of a selection of computation kernels that consume and produce elements from streams of data. The advantages of this decomposition are multi-fold. First, it separates communication (the gathers and scatters of data to and from global memory) from the actual computation. Hence, communication can be scheduled ahead of the corresponding computation, thereby hiding the cost of the large memory latency that is unavoidable in modern machines. Stream programs explicitly identify which variables are only names for values in a communication stream and don't need to be written back to memory, and which hold persistent application state. This information reduces the global memory bandwidth, another critical resource in a modern machine. In addition, the stream formulation allows the compiler to expose data-level parallelism (DLP) between stream elements in a kernel and thread-level parallelism (TLP) across kernels.

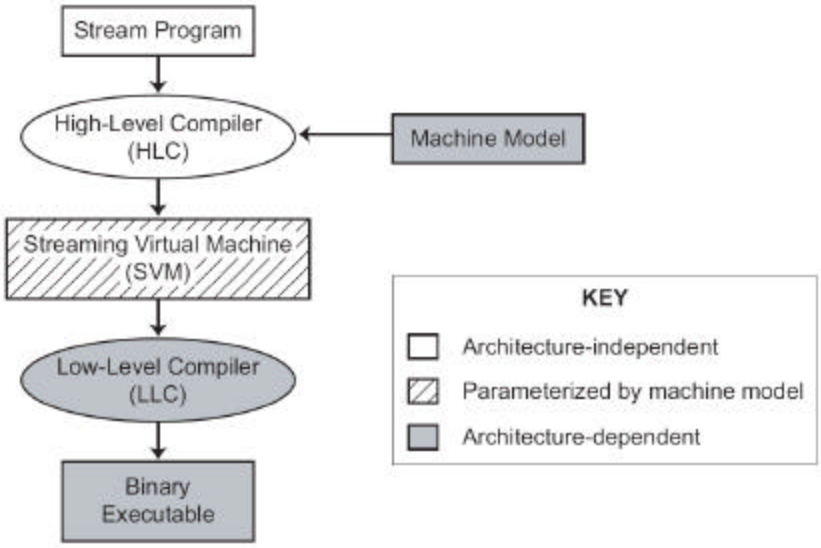

Figure 14 – Two level compiler scheme

The stream architecture space spans from configurable or statically scheduled grid multiprocessors like the MIT Raw project or the UT Austin TRIPS to SIMD stream coprocessors with a large local memory for stream buffering, Stanford's Imagine and

commodity graphics processors. Similar diversity exists with streaming programming languages. They vary from synchronous data-flow languages with infinite linear streams (StreamIt, Simulink), to languages with support for multi-dimensional streams and stencils (Brook), and to array languages (Matlab). The fragmentation in stream architectures and languages creates an interoperability problem that hinders the wide adoption of stream computing. To run any stream program on any stream architecture, one must develop a separate compiler for every language and architecture pair. The two-level compilation approach in Figure 14 can mitigate the engineering complexity of developing a new stream language or architecture. The high level compiler (HLC) is written once for each stream language and is responsible for parallelism detection, load balancing, coarse-grained scheduling of stream computations, and memory management for streaming data. The HLC inputs a stream program and targets an abstract architecture model. The abstract model is parameterized to describe basic topology and performance features of specific stream architectures. The low level compiler (LLC) is written once for each stream architecture and is responsible for instruction memory management and scheduling within each kernel. It inputs the abstract stream code and generates binary code. Apart from allowing for interoperability, this compilation model allows the language and HLC developers to focus on fundamental stream optimizations rather than over-specializing the compiler for the idiosyncratic features of any particular architecture.

4.2.1. The SVM Architecture Model

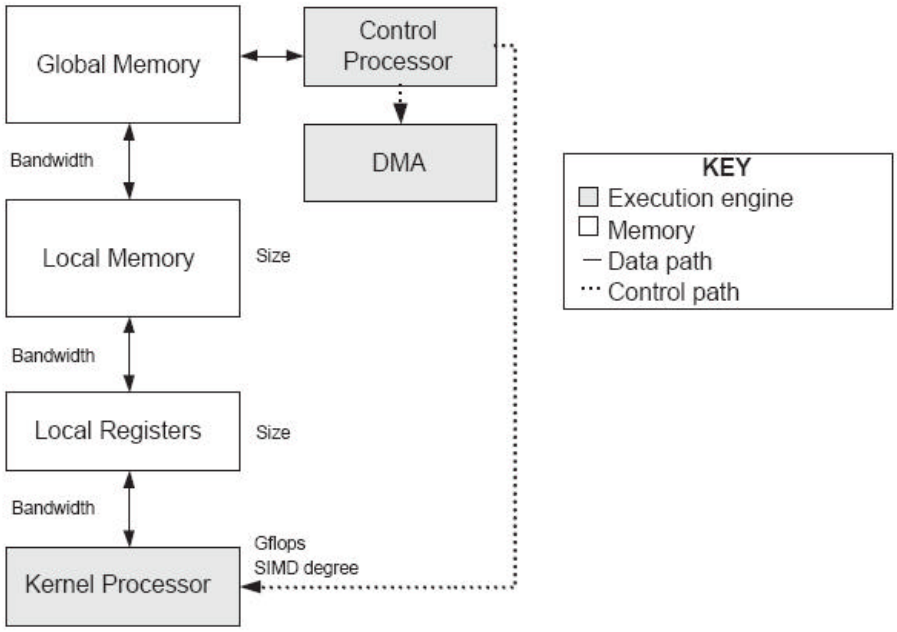

Figure 15 – SVM Architectural Model

The basic architecture model of the SVM shown in Figure 15 contains three different types of execution engines, each with its own thread of control. The Control Processor is the master processor and controls the operation of the entire machine. The control processor essentially issues all the operations that the machine executes. The Kernel and DMA engines simply allow the control processor to execute higher level operations. The kernel processor accelerates the computation of each of the stream kernels, while the DMA engine is used to manage the data movement that each kernel requires. By giving each of the processors its own thread of control, the SVM allows the control processor to run ahead of the actual data execution essentially prefetching future operations for the data and memory execution units. It also explicitly provides the dependence between all the stream operations to each of execution engines. Thus the execution engines are able to reorder computation for more efficient execution. Local memory provides the needed space to buffer data between different execution engines. For the SVM to work as a good parametrized architectural model, one must be able to abstract a wide class of stream machine into this model, while maintaining the ability to estimate the performance of the real machine. While clearly modeling the computational performance of the engines is important, it is also important to model the size and bandwidths of the memory in the machine. A SVM has three types of 5 components:

Processors can have multiple master processors (this is the case for stream processors and DMA engines). DMA engines can only run special kernels which only move data around; processors can run user-defined code. These processors capable of running user-defined code are then characterized by such factors as their operating frequency, mix of functional units, number of registers and SIMD level.

Memories come in three different flavors: FIFOs, RAMs and caches. All types are characterized by their size in bytes. RAMs are also defined by their coherence with regards to other memories in the system and the bandwidth for different types of accesses, namely sequential and random access.

Network Links connect one or many senders (processors, memories or network links) to one or many receivers. Each network link is characterized with a bandwidth and latency. Stream processors take advantage of high bandwidth local RAM memories or FIFOs that link stream processors together to reduce demands on global memory bandwidth through re-use and producer-consumer locality.

5. Software Tools for Checking Code

A non-trivial barrier to developing new hardware is the cost of simultaneously building the large software infrastructure needed to support it (compilers, operating systems, runtime systems). The cost of verifying this infrastructure can rival or even dwarf that needed to write it in the first place. Reducing this cost can go a long way to helping support lightweight hardware customization. To this end, we have focused on developing automatic techniques for finding errors of such code.

We use two main approaches: (1) customizable static analysis and (2) implementation-based software model checking. We have validated these methods on large, real, verification-unfriendly code bases (ranging from FLASH cache coherence software to Linux operating system code).

In terms of practical impact, our techniques have found thousands of errors in widely used open source software, many of which have been fixed in response to our reports. These tools are being commercialized in a company we started, Coverity, and have been licensed at roughly 20 large commercial sites.

We give an overview of our two main approaches below.

5.1. Customized static analysis: metacompilation (MC)

Systems software such as OS kernels, embedded systems, and libraries must obey many ad hoc, often obscure rules. Examples include "accesses to variable A must be guarded by lock B," "system calls must check user pointers for validity before using them," and "do not call blocking operations with interrupts disabled." A single rule violation can crash the system. These violations are often difficult to diagnose because of poor visibility (e.g., a locked up machine) or delayed effects (e.g., a crash from a missed interrupt re-enable many thousands of cycles previously).

We developed metacompilation (MC) as a general, lightweight method to find errors in such ad hoc rules. The key feature it exploits is that that many of the abstract properties relevant to rules map clearly to concrete code actions. We can thus use a compiler to check these actions for errors. For example, ordering rules such as "interrupts must be enabled after being disabled" reduce to observing the order of function calls or idiomatic sequences of statements (in this case, a call to a disable interrupt function must be followed by a re-enable call).

Of course, to check a rule, the compiler must first know it. Since many rules are domain or even system specific, hard wiring a fixed set into the compiler is ineffective. MC attacks this problem by making it easy for implementers to extend compilers with lightweight, system-specific checkers and optimizers. Because these extensions can be written by system implementers themselves, they can take into account the ad hoc (sometimes bizarre) semantics

of a system. Because they are compiler based, they also get the benefits of automatic static analysis. Unlike verification, compilers work with the code itself, removing the need to write and maintain a correct specification. Unlike testing, static analysis can examine all execution paths for errors, even in code that cannot be conveniently executed. Finally, all traditional approaches require effort proportional to the size of code. In contrast, once the fixed cost of writing an extension is paid, it can be run on 10 million lines as easily as 10 --- the main incremental cost is inspecting more errors.

We have developed a variety of tools based on this approach. All of them have found many errors in widely-used software, usually measured in hundreds.

- Metal: an extensible language and system for writing system-specific, static analyses [PLDI'02]. Metal checkers are simple (often less than 100 lines of code) yet find many serious errors in complex, real systems code. Further, is easily applied. Most our checkers were written by programmers who had only a passing familiarity with the systems that they checked.

- RacerX: a static tool for finding deadlocks and race conditions, which uses non-traditional methods to allow it to be quickly, effectively applied to large software systems. A novel feature of the checker is the use of statistical analysis to infer which locks protect which variables. It has found roughly 30 serious errors in a large commercial system, with a relatively low number of false positives [5].

- ARCHER: a tool that uses scalable path-sensitive, interprocedural analysis and symbolic constraint solving to find memory corruption bugs. It found hundreds in sendmail, postgres, Linux, BSD, etc. with less than a 30% false positive rate [6].

- MECA: an extensible annotation language and checking system for describing and enforcing security rules. It has a number of unusual features, including the ability to write programmatic annotators that mark large amounts of code automatically, and a statistical annotation propagation algorithm that gives us much more coverage than traditional approaches. It found over thirty security holes in Linux, despite running on code that had already been checked for these problems [7].

- MCjava: an extensible checking system for Java based on the ideas in Metal.

Recently, using static analysis to find program errors has become a very active topic. In addition to the tools above, we have also developed general techniques that other approaches should find useful.

- Z-ranking: uses statistical analysis to detect and counter when static analysis makes a mistake. The basic intuition: the most trustworthy analysis decisions are those that lead to many successful checks and relatively few errors. A given analysis mistake will

tend to flag many errors and have few successful checks. Z-ranking gives a way to rank such errors below those flagged by more trustworthy decisions, thereby reducing the impact of their false positives [8].

- F-ranking: a second, more generic way to reduce the impact of false positives. It uses the fact that both true error messages and false positives are highly clustered. For example, given a set of related messages (e.g., messages flagged by a checking tool in the same function or file) then if one is a true error (or a false positive) it is likely that the others are as well. We can use this correlation to demote or promote related messages during inspection. It can reduce the observed rate of false positives by a factor of 2 to 8 fold [9].

- Belief analysis. It turns out that a first order barrier to finding bugs is simply knowing what to check. Systems have hundreds to thousands of undocumented rules; leaving these unchecked means that most bugs in the system will not be detected. Belief analysis is a general technique that infers both rules and the state of the system by using programmer actions. For example, a dereference of a pointer, "p," implies a belief that "p" is non-null, a call to "unlock(l)" implies that "l" was locked, etc. Almost all of the checking tools described above use some aspect of this approach to broaden the properties they check [10].

5.2. Implementation-level Software Model Checking

While static checking finds many errors in rules that appear in the "surface" of code (e.g., that "lock()" is followed by "unlock()") it has difficulty checking properties implied by code (e.g., that a routing table has no loops, that a cache line has no sharers when it is in the exclusive state).

When applicable, formal verification methods can find such deep errors. One option is explicit model checking, which systematically enumerates the possible states of the system. A basic model checker starts from an initial state and recursively generates successive system states by executing the nondeterministic events of the system. States are stored in a hash table to ensure that each state is explored at most once. Checkpointing lets it do efficient systematic exploration of each action possible in a given state (file creation, deletion, crash, etc) by choosing one, doing it, checkpointing this resulting state, rolling back to the original state, doing the next operation, etc. Cannonicalization reduces semantically equivalent but superficially different states to the same state. This process continues either until the whole state space is explored, or until the model checker runs out of resources. When it works, this style of state graph exploration can achieve the effect of impractically massive testing by avoiding the redundancy that would occur in conventional testing.

The dominate cost of traditional model checking is the effort needed to write an abstract specification (commonly referred to as the “model”) of the system be provided. This upfront cost has traditionally made model checking completely impractical for large systems. A sufficiently detailed model can be as large as the checked system. Empirically, implementors often refuse to write them, those that are written have errors and, even if they do not, they “drift” as the implementation is modified but the model is not.

We have developed a new model "implementation-level" checker, CMC, that work with the implementation code directly without requiring an abstract specification. We have used it to successfully check real systems whose size and complexity would otherwise put them beyond reach of traditional model checking. We have also developed model checking techniques to handle the difficulty in checking systems orders of magnitude larger than what is typically attempted. We have used this to check three types of code (in order of increasing difficulty):

- AODV ad hoc network routing protocol. We checked three implementations of this protocol. We found 34 distinct errors, a rate of roughly one bug per 328 lines of code. Several bugs were non-trivial ones, difficult to find by any other method. In an ironic twist, model checking the implementation found a bug in the specification of AODV itself (this last error was confirmed by the authors of the AODV specification) [11].

- TCP. We checked the Linux TCP stack, which was roughly 10 times bigger than any AODV protocol. We found four errors. A surprising result was that it was easier to run the entire Linux kernel in the CMC model checker than extract out TCP in a stand-alone version [12].

- File system code. We extended the infrastructure used to check TCP to allow easy model checking of file system code. We then applied CMC to three widely-used, heavily-tested file systems, JFS, ReiserFS, and ext3. We have found serious bugs in all of them, 30 in total. Most have led to immediate file system patches (within a day of diagnosis). For each system, the model checker has found demonstrable events that lead to the unrecoverable destruction of metadata and entire directories. In several cases we found the misguided deletion of perfectly valid, long-lived, parent directories. In practice, several errors result in the deletion of all the contents of the file system root directory (i.e. "/") [13].

6. APPLICATIONS

To test out our architecture ideas we worked with a number of different applications, ranging from signal processing intensive tasks to much less regular thread applications.

6.1. Imagine Stream Applications

Imagine achieves 7.96 GFLOPS / 25.4 GOPS ALU performance on synthetic benchmarks and sustains approximately 50% on application benchmarks. Imagine can handle a wide range of applications including stereo depth extractor, video stream encoding, space-time adaptive processing, polygon rendering with OpenGL and Reyes, network packet processing, and wireless base station system. First, applications are enumerated with brief introduction and then key kernels of the applications are described.

6.1.1. Applications

DEPTH Details of stereo depth extractor are explained in Section 2.1. In Imagine processor running at 200 MHz, more than 90 frames of two 320x240 gray-scale images are processed per second.
MPEG This application encodes three frames of 360x288 24-bit video images according to the MPEG-2 standard. Imagine processor sustains a compression rate of 138 frames per second while consuming 6.8 Watts.
STAP Space-time adaptive processing [14] application spends most of run-time on one of its core components, QRD, which converts a 192x96 complex matrix into an upper triangular and an orthogonal matrix. Imagine processor decomposes the complex matrix 326 times per second.
RENDER A variety of polygon rendering applications are written including ones implementing traditional OpenGL rendering pipeline as well as Reyes rendering pipeline [15]. For the program rendering the first frame of the SPECviewperf 6.1.1 advanced visualizer benchmark using the Stanford Real-Time Shading Language [16], Imagine processor takes less than 23 ms.
WIRELESS This application implements wireless base-band algorithms including multi-user estimation, multi-user detection and Viterbi decoding for W-CDMA based cellular system. In simulation running at 500 MHz with one more multiplier per cluster, it performs 48x better for channel estimation and 42x for detection, against TI C67 DSP [17].
PACKET High-speed programmable Packet Processing/Inspection [18, 19]. Imagine achieves throughput of 2.9 Gb/s for IPv4 forwarding and 1.6 Gb/s for AES encryption in simulation.

6.1.2. Kernels

Table 3 : Key kernels of Imagine applications

Application	Kernel	Description
DEPTH	blockfill	fills a stream with a constant value
	blocksad	sum of absolute difference over a sliding window
	byte2word	unpacks 8-bit pixels into 16-bit pixels
	convfx3x3	convolves a row with a 3x3 filter
	convfx7x7	convolves a row with a 7x7 filter
	exdepth	extracts depth from SAD values
	extemp3	handles final rows for 3x3 convolution
	extemp7	handles final rows fro 7x7 convolution
MPEG	blocksearch	searches reference image to determine motion vector
	corr	correlates current macroblock with reference macroblock
	dct	discrete cosine transform
	diff	computes the difference between two macroblocks
	icolor	color space conversion from RGB to YCrCb
	idct	inverse discrete cosine transform
	idxgen	generate addresses to access a macroblock
	mv2idx	converts a motion vector into addresses
	pcolor	color space conversion from RGB to YCrCb
	rle	run-length encodes a macroblock
STAP	house	Householder transformation
	update1	updates matrix
	update2	updates matrix
RENDER	compact_recycl	compacts conflicting fragments
	e	computes shading and lighting values
	glshader	hashes conflicting fragments
	hash	merges two streams of sorted fragments
	mergefrag	perspective projection
	project	sorts groups of 32 fragments
	sort32frag	converts a span into fragments
	spanrast	converts a triangle into spans
	spansgen	prepares a triangle for span generation
	spansprep	transforms a triangle from object space to screen space
	xform	compares z values of two streams of fragments
	zcompare	

WIRELESS	matrix_mul iteration mf pic viterbi	matrix-matrix multiplication kernel iteration update kernel matched filter kernel cancels interference from other users to make a better estimate viterbi decoder to decode 32 users
PACKET	key_expansion core final_round	AES key scheduler algorithm block encryption rotate and mask for the byte substitution transformation

6.2. Parallel Programming with TLS

We have also investigated exposing the TLS to the programmer in the context of manually parallelizing general purpose applications from the SPEC CPU2000 benchmarks which are written in C. Using source code transformations that are beyond the capabilities of automatic parallelizers we have demonstrated good speedups on floating point applications and integer applications. To demonstrate the best methods for manual TLS parallelization, we created a small C benchmark that is difficult to parallelize automatically. We used this benchmark to show methods of extracting parallelism that can only be done manually. Similarly, to clarify the effectiveness of each method and the speedup obtainable only via manual parallelization, we separated the speedups on the SPEC CPU2000 floating point applications into the incremental speedup derived from the use of each method. Similar work was performed for the integer benchmarks. This effort represents the the most extensive attempts that we are aware of to manually parallelize the SPEC200 CPU benchmarks.

6.3. Race Trace and Shading Language

Two application areas were investigated in depth as part of the Smart Memory Project: Ray Tracing and Shading Languages.

6.3.1. Ray Tracing Architectures [20-24]

Today's rendering hardware is capable of rendering millions of polygons per second. Advances in hardware should push this number toward billions of polygons per second in the next decade. This speed would at first glance seem to be enough to render any scene. However, the shading models currently used are inadequate. Any attempt to add specularity or shadows is done as a hack. These hacks require multiple passes through scene data. As scenes grow in complexity, multiple passes through the data becomes less and less desirable. Aside from offering more realistic surface shading, ray tracing offers implicit occlusion culling, and for large scenes can exhibit better memory coherence. For these reasons, it has often been suggested that ray tracing will eventually be faster than current scanline rendering.

Ray tracing was also a major motivating problem in the Data Intensive Systems Program. The program XPATCH uses ray tracing to compute radar cross-sections of complex models. We have looked into the DIS ray tracing benchmark and have found many opportunities for improvement. Ray tracing is also used for Computer-Generated Forces to support simulation and training. For example, entities employ ray tracing for line-of-sight calculations; that is, to check if they see each other.

As part of the Smart Memory Project, we built a clean, compact ray tracing system with OpenGL style lighting and shading. This software was used to evaluate the potential of ray tracing algorithms in hardware and compare it to standard scanline implementations. The goal is to show under which circumstances ray tracing will be faster, more cost effective, and more flexible than current graphics hardware.

Two ray tracers were being produced for this subproject:

The first is a simple, single threaded ray tracer written as cleanly and concisely as possible. This ray tracer has all the functional components of the proposed architecture, but will emphasize clarity of design over performance. This software will be used primarily for conceptual discussions. The goal was to have a simple system in place for discussing essential ray tracing algorithms without heavy optimizations and unnecessary code confusion. For simplicity's sake, this ray tracer assumes the entire scene (including textures) will fit into main memory.

The second ray tracer was a software implementation of the proposed architecture. This software will be used to study various aspects of the architecture like memory bandwidth (scene DB and Texture DB), ray bandwidth, intersection bandwidth, intersection time (grid and triangle), and shading time. This data can then be used to improve the current software design, and find areas where hardware components could be made to push the system towards interactivity.

In a nutshell, the system consists of many different data managers, databases, and functional units, all organized in a feedback loop. The basic path of a ray is to first get scheduled to be traced, second get intersected with both the acceleration structure and then the geometry, third get shaded, and fourth report shading evaluations to the final image. This loop contains feedback for secondary (shadow, reflection, refraction) rays generated from shading to be themselves shaded.

We built a simple ray tracer with all the major functional units of the proposed architecture. This ray tracer was designed to be a testbed for core algorithms of the architecture without being unnecessarily complicated with optimizations and memory concerns. It was intended to be the common frame of reference between people familiar with ray tracing and those unfamiliar with the core workings. It was to be used to try out various algorithms, and to

eventually compare its performance to the more complex ray tracer. All of the data structures and basic algorithms used in the simple ray tracer are the same as those proposed for the complex architecture. The algorithms for grid traversal, triangle intersection, ray queuing, and pixel shading are in place, though not in any speed optimized form.

The simple ray tracer was used to derive various statistical properties of the ray tracing process that help us refine and optimize the main ray tracing architecture. To that end we identified key statistical values for evaluating ray tracing architectures. The simple ray tracer produced statistical information on each traced scene including: number of rays generated, number of voxels a ray hits, number of primitives a ray misses before a hit, number of rays passing through a given voxel, number of primitives in a voxel, number of voxels a primitive spans, number of ray/triangle intersections, number of ray/triangle hits and misses, plus histograms of nearly all this data for analysis of distribution patterns.

This basic ray tracing architecture was then modified to run on a stream processor. Each module was modified to run as a kernel program. The basic kernels were the eye-ray generator, the ray-traversal kernel, the ray-triangle intersect kernel, and the shading kernel. Two versions of the streaming ray tracer were implemented. One was implemented in the Brook programming environment, and the other was implemented in Cg, a commercial programming environment for GPUs. The Cg version was run on ATI and NVIDIA hardware, and demonstrated at SIGGRAPH 2002.

The results of the streaming ray tracer on GPUs were very encouraging. The basic ray-triangle intersect kernel was able to intersect over 100 million rays with triangles per second. This was faster than a highly optimized ray-triangle intersector for a Pentium CPU running at 800 Mhz (a 2002 class processor).

Subsequently, we showed how a more complicated ray tracing algorithm could be implemented on top of the basic ray-tracing framework. In 2003, we described a complete implementation of photon mapping running on the GPU. The photon mapper include new modules for tracing photons from the light sources and building a grid-based spatial indexing data structure that supported k-nearest-neighbor queries. This version required more advanced stream programming operators, in particular, scatter.
[1]
[2]

6.4. Real-time Shading Language [25-29]

Real-time graphics hardware is capable of rendering images using advanced texturing, shading, and lighting models. However, it is time-consuming and difficult to implement complex rendering algorithms on this hardware, because the programmer must configure the hardware at a low level. We have built a real-time system with a programmable-shading language that provides an abstraction layer between the programmer and the hardware. The

most important feature of our shading language is its capability to specify calculations at vertices or fragments. In our system, per-vertex calculations are implemented either on the CPU or GPU.

We have implemented two versions of our programmable shading system. The first version investigated fragment programmability by mapping simple per-fragment expressions to the OpenGL pipeline using multipass rendering. The second version added programmability at multiple computation frequencies, most importantly per-vertex, by mapping non-per-fragment computations to a host-side processor. The second version ran on both ATI and NVIDIA hardware, and used both register-combiner and vertex/pixel shader instruction sets.

The most ambitious compiler was the register combiner compiler. Register combiners are a data-flow architecture for fragment calculations. Each combining stage uses a VLIW architecture. The compiler was used to choose and schedule instructions for multiple register combiner units organized in a pipeline.

Another technology that was developed as part of this project was the ability to partition shaders into multiple passes. The input to the algorithm was a directed acycle graph representing the expression computed by the shader. Since the resources in the graphics hardware are limited, the expression often cannot be computed directly on the hardware. We developed the Recursive Denominator Split (RDS) algorithm to partition shaders close to optimally.

The system, RTSL, was released and is available in source form. The system had over 10,000 downloads in the first year (we are not currently maintaining the system). The technology for several of the compiler back-ends was licensed to NVIDIA and became part of their Cg language for GPUs. W. Mark, who was the main contributor to RTSL, was the Chief Architect of Cg. Shading languages have rapidly caught on and are now part of the everyday programming environment for graphics chips.

References:

[3] Takeo Kanade, Atsushi Yoshida, Kazuo Oda, Hiroshi Kano, and Masaya Tanaka. A stereo machine for video-rate dense depth mapping and its new applications. In *Proceedings of the 15th Computer Vision and Pattern Recognition Conference*, pages 196–202, San Francisco, CA, June 18–20, 1996.

[4] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, M. Horowitz. "Smart Memories: A Modular Reconfigurable Architecture", *ISCA*, June 2000.

[5] K. Mai, R. Ho, E. Alon, D. Liu, Y. Kim, D. Patil, and M. Horowitz. "Architecture and Circuit Techniques for a Reconfigurable Memory Block", *ISSCC*, February 2004.

[6]

[7] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Jin Namkoong, John D. Owens, Brian Towles, and Andrew Chang. Imagine: Media processing with streams. *IEEE Micro*, pages 35–46, Mar/Apr 2001.

[8] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. SOSP 2003.

[9]

[10] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. FSE 2003.

[11] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. MECA: an Extensible, Expressive System and Language for Statically Checking Security Properties. Proceedings of the 10th ACM conference on Computer and communication security (ACM CCS), 2003.

[12] Ted Kremenek and Dawson Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. SAS 2003.

[13] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation Exploitation in Error Ranking. To appear: Foundations of Software Engineering (FSE), 2004.

[14] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code, Appeared in SOSP 01.

[15] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, David L. Dill. CMC: A pragmatic approach to model checking real code OSDI 2002.

[16] Dawson Engler and Madanlal Musuvathi.Model-checking large network protocol implementations, NSDI 2004.

[17] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi, Using Model checking to Find Serious File System Errors, to appear, 6th Symposium on Operating Systems Design and Implementation (OSD '04)

[18] Kenneth C. Cain, Jose A. Torres, and Ronald T. Williams. RT STAP: Real-time space-time adaptive processing benchmark. Technical Report MTR 96B0000021, MITRE, February 1997

[19] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon rendering on a stream architecture. In *Proceedings of the 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 23–32, August 2000.

[20] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH*, pages 159–170, August 2001.

[21] Sridhar Rajagopal, Scott Rixner, and Joseph R. Cavallaro. A programmable baseband processor design for software defined radios. In *45th IEEE International Midwest Symposium on Circuits and Systems*, volume 3, pages 413–416, August 2002.

[22] Jathin S. Rai, Yu-Kuen Lai, and Gregory T. Byrd. Packet processing on a simd stream processor. In *Workshop on Network Processors & Applications - NP3*, February 2004.

[23] Jathin Sanoor Rai. A feasibility study on the application of stream architectures for packet processing applications. Master's thesis, North Carolina State University, Raleigh, NC, 2003.

[24] T, Purcell, I. Buck, W. Mark, P. Hanrahan, Ray-tracing on programmable graphics hardware, ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002), 21(3), 703-712, 2002.

[25] T. Purcell, In Chalmers et al., Parallel ray tracing on a chip, Ed., Practical Parallel Rendering, A. K. Peters, 2002.

[26] T. Purcell, C. Donner, M. Cammarano, H. Jensen, P. Hanrahan, Photon-mapping on programmable graphics hardware, Proceedings of the Eurographics/SIGGRAPH Symposium on Graphics Hardware, pp. 41-50, 2003 (Best Paper Award).

[27] I. Wald, T. Purcell, J. Schmittler, C. Benthin, P. Slussalek, Real-time ray tracing and its use for interactive global illumination, Eurographics State-of-the-art Report, 2003.

[28] T. Purcell, Stanford Ph.D. thesis ,Ray-tracing on a stream processor , March 2004.

[29] K. Proudfoot, W. Mark, S. Tzvetkov, P. Hanrahan, A real-time procedural shading system for programmable graphics hardware, Proceedings of SIGGRAPH 2001.

[30] W. Mark and K. Proudfoot, Compiling to a VLIW fragment pipeline, Proceedings of the Eurographics/SIGGRAPH Symposium on Graphics Hardware, 2001

[31] W. Mark, K. Proudfoot, The F-buffer: A rasterization-order FIFO buffer for multi-pass rendering, Proceedings of the Eurographics/SIGGRAPH Symposium on Graphics Hardware, 2001.

[32] E. Chan, R. Ng, P. Sen, K. Proudfoot, P. Hanrahan, Efficient partitioning of shaders for multi-pass rendering on programmable graphics hardware, Proceedings of the Eurographics/SIGGRAPH Symposium on Graphics Hardware, 2002 (Best Paper Award).

[33] W. Mark, Real-time programmable shading, In Procedural Texturing and Modeling: A Procedural Approach, D. Ebert et al., ed., 2003.

DISTRIBUTION LIST

DTIC/OCP
8725 John J. Kingman Rd, Suite 0944
Ft Belvoir, VA 22060-6218 1 cy

AFRL/VSIL
Kirtland AFB, NM 87117-5776 1 cy

AFRL/VSIH
Kirtland AFB, NM 87117-5776 1 cy

Stanford University
Gates 306, 353 Serra Mall
Stanford, CA 94305-9030 1 cy

Official Record Copy
AFRL/VSSE, Dr. Jim Lyke 1 cy